

Titolo: Introduzione alla programmazione Bash

Autore: T4n|n0 Ru|3z

Data: 13/04/2010

Website: www.chimerarevo.com

Contact: tanino_rul3z@hotmail.com

Note dell'autore

Vi sarei grato se mi segnalaste eventuali errori semantici o sintattici sia del linguaggio bash che del linguaggio italiano :) Questa guida ha solo scopo didattico e informativo. L'autore autorizza chiunque voglia modificarla e/o pubblicarla a citare sempre la fonte e l'autore originale.

Iniziamo!

Oggi vi parlerò un po' della shell di Linux e di programmazione Bash. Questa guida tratta argomenti basilari per chi vuole iniziare a concepire la programmazione bash e alcuni comandi utili di linux. Se volete approfondire ricordate che google è il vostro migliore amico.

INTRODUZIONE

Iniziamo con la definizione dell'acronimo **Bash** che sta per **Bourne Again Shell**. La shell non è altro che un interfaccia (testuale) che l'utente ha con il sistema operativo. Viene detta anche "interprete dei comandi" perchè è appunto un programma che interpreta gli input e i comandi passati dall'utente. La shell entra in funzione appena facciamo il login sulla nostra macchina e termina il suo lavoro al logout. Alcuni tipi di shell sono: *sh, tcsh, ksh*.

BREVE STORIA

La shell fu sviluppata nel a partire dal 1988 nel progetto GNU che in breve possiamo schematizzare così:

- Sviluppare una versione gratuita degli Unix
- GNU = Gnu's Not Linux
- Software libero, open ovvero FSF (Free Software Foundation)
- Software protetto da "copyleft" ovvero un software con questa licenza è distribuito gratis e con il relativo codice sorgente e deve essere mantenuto tale (non si può vendere software preso gratis).

Una delle prime shell fu scritta da Steve Bourne nel 1979 ed era la sh (ecco perchè l'acronimo Bourne Again Shell).

COMPITI DELLA SHELL

Proviamo a dare da shell il seguente comando:

```
bash> sort -n numeri > numeri.ordinati
```

ATTENZIONE: `bash>` indica che è 1 comando `bash`, dovete cominciare a dare il comando da dopo `bash>`, tutto ciò che è in corsivo per capirci.

Esaminiamo l'esempio: "numeri" è un nostro file creato precedentemente che contiene dei numeri scritti in modo disordinato. Con il comando `sort` noi andiamo ad ordinare il contenuto e lo mettiamo nel file *numeri.ordinati*.

La shell separa i token in:

```
sort, -n, num_telefoni, >, num_tel.ordinati
```

Vediamo nei dettagli i comandi e il loro significato:

- `sort` = è il comando da eseguire
- `-n` e `numeri` = sono gli argomenti di `sort`
- `>` = questo simbolo di maggiore è importante, è il simbolo di ridirezione, ovvero ci permette di ridirigere il nostro output su qualche altra parte, nel nostro caso sul file `numeri.ordinati`
- `numeri.ordinati` = nome del file di ridirezione

COMANDI ARGOMENTI E OPZIONI

Vediamo ora 1 altro esempio:

```
bash> lp -d lp1 file1 file2 file3
```

La linea dei comandi sono tutte le parole (stringhe) separati da spazi o TAB. Solitamente la prima parola è sempre il comando e il resto sono gli argomenti. Nel nostro caso il comando è `lp`. Una opzione invece e' un argomento speciale che impartisce istruzioni al comando, cioè' modifica il comportamento di default. Può capitare a volte che un opzione ha un proprio argomento.

PS: il comando `lp` serve a stampare (nel senso vero della parola, con una stampante :))

Altri comandi utili sono :

- **cwd**(Current Working Directory): ci dice in che directory ci troviamo
- **pwd**: identica a `cwd`, solo che funge al login

- **cd:** =serve per cambiare directory
- **tilde(~):** home directory
- **~user:** home directory dell'utente "user"

FILENAME E WILDCARD

Il wildcard permette di specificare più file. Le sue opzioni sono:

- **?** qualsiasi carattere (1 solo)
- ***** qualsiasi sequenza di caratteri (0 o +)
- **[set]** qualsiasi carattere in set
- **[!set]** qualsiasi carattere non in set

Ecco alcuni esempi:

- ***.txt** tutti i file che finiscono in .txt
- **p?ppa** pippa, poppa, pappa, plppa,
- ***.t[xyw]t** file che finiscono in txt, tyt, twt
- **[!abc]*** file che non iniziano con a b o c
- ***[0-9][0-9]** file che finiscono con 2 cifre
- **[!a-z]*** file che non iniziano con una lettera
- **?, *, !, []** sono caratteri speciali

RIDIREZIONE

Ho già precedentemente accennato al concetto di ridirezione, ora lo approfondiremo con alcuni esempi:

```
bash> cat < file1 > file2
```

Bisogna prima spendere due parole però sugli **standard error, input e output** che su molti sistemi vengono espressi rispettivamente con il codice **2,0,1**. Lo standard input (**0** o **STDIN_FILENO** per gli amanti del C) è l'input preso da tastiera, mentre lo standard output (**1** o **STDOUT_FILENO** per gli amanti del C) è il monitor.

Tutti questi standard se siete amanti del C li trovate nella libreria `unistd.h` :)

Il comando usato nell'esempio precedente è **cat**. Esso normalmente prende input da standard input e stampa su standard output, quindi serve a visualizzare il contenuto di un qualcosa. Ma nel nostro esempio, cosa succede?

Il simbolo minore è anch'esso simbolo di ridirezione ma questa volta dell'input, infatti diciamo a `cat` di prendere l'input da `file1` e non da tastiera.

L'output è anch'esso ridirezionato e quindi il risultato sarà nel file2 (nel caso specifico è un pò un esempio stupido,infatti si copia il contenuto di file1 in file2 :D)

Ricapitolando tutto abbiamo:

- `< file1` fa si' che l'input sia preso da file1
- `> file2` fa si' che l'output vada nel file2

PIPELINE

La pipeline è molto importante e utile. Essa serve a mandare l'output di un programma nell'input di un altro programma.

Introduciamo anche il comando `cut` che serve a selezionare una parte di ogni riga di un flusso di testo e a inviarla sullo standard output.

Diamo il seguente comando:

```
bash> cut -d: -f1 /etc/passwd | sort | lp
```

il file `/etc/passwd` contiene tutte le password di sistema; cut nel nostro caso prende il primo campo `-f1` separato dal delimitatore `-d`.

PROCESSI IN BACKGROUND

Un processo è un programma in esecuzione. Se eseguiamo un comando da shell, dobbiamo aspettare che esso termini. Volendo però, possiamo far passare anche il processo in background grazie al simbolo `&`.

```
bash> tar -xzf archivio.tgz &  
[1] 3156  
bash>
```

nel nostro caso metteremo in background l'estrazione dei file da `archivio.tgz` e se vogliamo sapere qualche informazione aggiuntiva sui processi in background possiamo digitare `jobs`. Il valore 3156 corrisponde al `pid` del processo mentre [1] si riferisce al numero di job (ne abbiamo solo uno di job in questo caso).

`jobs` (informazioni sui processi che sono in esecuzione)

- `-l` stampa anche i process ID
- `-p` stampa solo i process ID
- `-s` stampa solo i processi in stato di Stop. Es: attesa

- di input dalla tastiera
- -r stampa solo i processi in in stato di Running

A fine operazione avremo:

```
bash>  
[1]+ Done tar -xzvf archivio.tgz
```

Bisogna spendere però altre due parole sui processi in background. Essi infatti non dovrebbero avere I/O. Il terminale è uno solo e solo un processo può usufruirne: in questo caso parliamo di **foreground**.

Cosa capita se un processo in background ha I/O ?? Semplice: se vuole input si blocca aspettandolo oppure se ha output esso viene mescolato sul video con quello che stavamo facendo.

Noi però siamo abbastanza bravi da capire che possiamo ovviare al problema con le ridirezioni :D

CARATTERI SPECIALI

Alcuni caratteri hanno significati speciali :

- ~ home directory
- # commento
- \$ precede il nome di variabile
- & processo in background
- ? * [] per wildcard (nomi file)
- | pipe
- () inizio e fine subshell
- { } blocco di comandi
- ; separatore di comandi
- ' quote (virgoletta) - *forte*
- " quote (virgolette) - *debole*
- < > ridirezione
- ! simbolo di negazione
- / (slash) separatore directory nel nome del file
- \ (backslash) simbolo di "escape"

Facciamo alcuni esempi della potenza di quoting (virgolette)

```
bash> echo 2 * 3 > 5 espressione vera  
bash> _
```

cosa succederà? La shell interpreterà i comandi e ci creerà un file di nome "5" con dentro "2", seguito dai nomi dei file nella cwd (*) e da "3 espressione vera". Non ci credete? Provare per

credere :P

Il nostro scopo è invece quello di stampare a video ciò che passiamo a echo:

```
bash> echo '2 * 3 > 5 espressione vera'  
2 * 3 > 5 espressione vera  
bash> echo "2 * 3 > 5 espressione vera"  
2 * 3 > 5 espressione vera
```

Possiamo anche avvalerci dell'aiuto del backslash escaping (che toglie il significato speciale):

```
bash> echo 2 \  
2 * 3 > 5 espressione vera
```

Bisogna stare attenti a quando però usiamo il backslash con i quote deboli:

```
bash> echo 'L\'\'aquila non volava.'  
L'aquila non volava.
```

CARATTERI DI CONTROLLO

Tasto **CTRL** + un'altro tasto, ad esempio:

- **CTRL-C** interrompe il processo (SIGINT)
- **CTRL-** interrompe il processo (SIGQUIT)
- **CTRL-Z** sospende il processo (SIGTSTP)
- **CTRL-D** fine input (EOF)
- **CTRL-S** sospende output video
- **CTRL-Q** ripristina output video

Questi sono le combinazioni più famose, conosciute e di default; se invece vogliamo maggiori info sui caratteri di controllo del nostro sistema possiamo digitare il comando **stty -a**.

Editor di linea

I due editor più famosi sono **vi** ed **emacs**. Per rendere uno degli editor di linea quello default si possono usare i comandi:

- `set -o emacs`
- `set -o vi`

Sinceramente non vi spiego nei dettagli l'utilizzo dei due editor, la rete è piena di manuali molto ben dettagliati e completi :)

History

Il comando `history` ci permette di visualizzare la storia di tutti i comandi che abbiamo digitato nella nostra shell. Esistono alcuni comandi per fare delle ricerche nella nostra history e ve li schematizzo in modo semplice e veloce:

- `!` inizia una ricerca nella "history"
- `!!` esegue il comando precedente
- `!47` esegue il 47-esimo comando
- `!-23` comando corrente - 23
- `!str` ultimo comando che inizia per str
- `!?str?` ultimo comando che contiene str
- `^s1^s2^` ultimo comando, sostituisce s1 con s2

Possiamo associare questi comandi con i token:

- `n` (n+1)-esima parola (0 = prima)
- `^` primo argomento (seconda parola)
- `$` ultimo argomento

Ecco alcuni esempi:

- `!!:0` prima parola del comando precedente
- `!!:` ultima parola del comando precedente
- `!!:3-6` dalla 4a alla 7a parola
- `!!:*` tutte le parole tranne la prima
- `!!:2-*` dalla 3a all'ultima parola

ALCUNI FILE SPECIALI

Esistono alcuni file che hanno una particolare importanza e sono:

- **`.bash_profile`** (file che viene eseguito ad ogni login e qualsiasi modifica ad esso ha effetto al prossimo login. Deve contenere quindi comandi che devono essere eseguiti solo al login)
- **`.bashrc`** (contiene tutti i comandi che vogliamo in ogni subshell)
- **`.bash_logout`** (viene eseguito al logout)

Non aggiungo altro per questi file perchè in rete troverete tanti riferimenti e approfondimenti se volete saperne qualcosa in più.

IL COMANDO ALIAS

Il comando `alias` ci permette di ridefinire il nome e la funzione di comandi originali, oppure di crearne di nuovi più sofisticati. La sua sintassi è

`alias nome=comando`

Vediamo alcuni esempi:

```
alias If='ls -F'
```

abbiamo creato un comando IF che seguirà sempre “ls -F” oppure

```
alias shut='shutdown -t15 -hfn now'
```

questo fa sì che abbiate creato il nuovo comando shut, che esegue uno shutdown della macchina.

Attenzione: Non si possono usare parametri. Esistono le funzioni per questo.

LE VARIABILI

La sintassi per definire una variabile è:

nome=valore

Non ci deve essere spazio intorno al segno di “uguale” (=) e se **valore** contiene degli spazi, allora bisogna usare i doppiapici nel seguente modo: “valore”.

Le variabili vengono usate attraverso la seguente logica:

\${nome}

Le parentesi graffe {} possono essere anche omesse se non c'è ambiguità.

“A differenza di molti altri linguaggi di programmazione, Bash non differenzia le sue variabili per “tipo”. Essenzialmente le variabili Bash sono stringhe di caratteri, ma in base al contesto, la shell consente le operazioni con interi e i confronti di variabili. Il fattore determinante è se il valore di una variabile sia formato, o meno, solo da cifre.

Le variabili non tipizzate sono sia una benedizione che una calamità.

Permettono maggiore flessibilità nello scripting (abbastanza corda per impiccarvici!) e rendono più semplice sfornare righe di codice; come “contro” invece, consentono errori subdoli e incoraggiano stili di programmazione disordinati.

È compito del programmatore tenere traccia dei tipi di variabili contenute nello script. Bash non lo farà per lui. “

(da www.pluto.it)

Per cancellare una variabile si usa invece la sintassi:

unset nome

Se una variabile che non e' definita vale "" (stringa nulla) e con l'opzione nounset, l'uso di una variabile non definita causa un errore.

VARIABILI D'AMBIENTE

Alcune variabili d'ambiente già definite e predefinite (ma che possiamo cambiare) sono :

- **HOME:** home directory
- **PATH:** ricerca dei comandi
- **EDITOR:** vi o emacs (normalmente)
- **PS1:** prompt
- **HISTFILE:** nome del file per l'history
- **PWD:** current working directory (path)
- **OLDPWD:** directory precedente
- **SECONDS:** contati dall'inizio della shell
- **IFS:** separatori (internal field separator)

Prendiamo come esempio il prompt. Se diamo una cosa del genere:

```
PS1="[ \u@\h]> "
```

avremo come output:

```
nostrusername@hostname>
```

Possiamo personalizzare il prompt:

- **\d** data
- **\h,\H** hostname
- **\s** nome della shell
- **\t,\T** orario
- **\u** nome utente
- **\v,\V** versione di bash
- **\w,\W** cwd
- **!\,\#** il numero del comando (history)
- **** backslash
- **\nnn** carattere in ottale

Di default comando = una linea.

Se vogliamo usare più linee basta usare \ per indicare che si vuole continuare sulla linea successiva.

Se ci sono delle virgolette aperte invece è tutto automatico.

Vediamo un esempio:

```
bash> echo comando che \  
>usa due linee.  
comando che usa due linee.  
bash> echo "comando che  
>usa due linee."  
comando che  
usa due linee.  
bash>
```

I processi che hanno come padre la shell possono accedere alle variabili di ambiente.

Le altre variabili devono essere "esportate" altrimenti non sono visibili. Si usa in tal caso la sintassi:

- **export nome**
- **export nome =valore**

SCRIPT

Vediamo un esempio di semplice script bash:

```
#!/bin/bash  
echo Questo e' un script  
echo I file in /bin sono:  
ls /bin
```

Uno script è un file con comandi shell. Per eseguirlo possiamo dare il comando:

source filename

oppure

./filename

ma potrebbe essere necessario dare prima i permessi di esecuzione con

chmod +x filename

Il secondo metodo lancia una subshell e lancia lo script nella subshell.

La prima riga che inizia con **#!/** indica sostanzialmente

#!/comando,opzionale

dove "comando" viene usato per interpretare lo script.

FUNZIONI

Le funzioni bash hanno lo stesso concetto di tutte le funzioni dei linguaggi di programmazione. Esse possono essere definite anche fuori dallo script.

Ecco una sintassi generale di una funzione:

```
function nome {
    comandi shell
}
nome () {
    comandi shell
}
```

Con il comando **declare -f** abbiamo una lista delle funzioni definite.

PARAMETRI

I parametri sono passati in variabili speciali. Valgono sia per lo script che per le funzioni:

`$1`, `$2`, ..., `${10}`, `${11}`, ...

`$0` e' il nome dello script.

`$*` Rappresenta l'insieme di tutti i parametri posizionali a partire dal primo.

Quando viene utilizzato all'interno di apici doppi, rappresenta un'unica parola composta dal contenuto dei parametri posizionali, spaziatosi dal primo carattere contenuto nella variabile speciale `'IFS'`. Se questa variabile non è definita, oppure contiene una stringa vuota, viene utilizzato uno spazio singolo.

Per esempio, se ``IFS'` contenesse la sequenza `'xyz'`, `""$*"` sarebbe equivalente a:

```
""$1x$2x..."
```

La variabile di shell `'IFS'` contiene di solito la sequenza: `<SP><HT><LF>` (corrispondente a uno spazio normale, un carattere di tabulazione e al codice di interruzione di riga nella maggior parte dei sistemi Unix). Di conseguenza, viene utilizzato normalmente il carattere spazio (`<SP>`) per staccare i vari parametri posizionali. Per cui, in pratica, la maggior parte delle volte, `""$*"` equivale a:

```
""$1 $2..."
```

\$@ Rappresenta l'insieme di tutti i parametri posizionali a partire dal primo.

Quando viene utilizzato all'interno di apici doppi, rappresenta una serie di parole, ognuna composta dal contenuto del parametro posizionale rispettivo. Di conseguenza, `"$@"` equivale a `"$1" "$2"`.

Questo rappresenta un'eccezione rispetto agli altri parametri: l'espansione di questo parametro genera diverse parole.

\$# Restituisce il numero della quantità di parametri posizionali.

\$? Restituisce lo stato dell'ultima pipeline eseguita in primo piano (foreground).

\$- Il trattino, restituisce la serie di lettere corrispondenti alle modalità configurabili attraverso il comando interno ``set'`.

\$\$ Restituisce il PID della shell. Se viene utilizzato all'interno di una subshell, cioè tra parentesi tonde, restituisce il PID della shell principale e non quello della subshell.

\$! Restituisce il valore dell'errore.

\$_ Il simbolo di sottolineatura (underscore), restituisce l'ultimo argomento del comando precedente.

VARIABILI LOCALI E GLOBALI

Vediamo questo script bash:

```
function funzione1 { # questa funzione richiede due parametri
    echo funzione $0: $1 $2
    pippo="ciao dalla funzione1"
    echo pippo = $pippo
}
```

```
pippo="ciao dallo script"
echo pippo = $pippo
echo script $0: $1 $2
funzione1 parametro1 parametro2
echo sono tornato nello script
echo pippo = $pippo
echo script $0: $1 $2
```

Cosa succede se lo eseguiamo?

```
bash> ./scriptfile arg1 arg2
pippo = ciao dallo script
script ./scriptfile: arg1 arg2
funzione ./scriptfile: parametro1 parametro2
pippo = ciao dalla funzione1
sono tornato nello script
pippo = ciao dalla funzione1
script ./scriptfile: arg1 arg2
```

Notiamo subito una cosa: in uno script le variabili sono sempre globali anche se definite in delle funzioni. Possiamo definirle locali invece all'inizio della funzione con:

```
local pippo
```

OPERATORI DI STRINGHE

Sono molto utili soprattutto per manipolare nomi di file. Esempio: cambiare il nome da .gif a .jpg

```
nuovonome=${vecchionome%.gif}.jpg
```

ATTENZIONE AGLI SPAZI:

- `${var}`
- `${var:offset}`

vanno scritti senza spazi bianchi.

Ecco la lista:

- `${var:-stringa}` se var non è nulla vale `${var}`, altrimenti vale stringa.
- `${var:=stringa}` se var non è nulla vale `${var}`, altrimenti vale stringa e stringa viene assegnata a var.
- `${var:?messaggio}` se var non è nulla vale `${var}`, altrimenti viene stampato il messaggio e lo script termina.
- `${var:+stringa}` se var non è nulla vale stringa, altrimenti vale NULL
- `${var:offset}` sottostringa di `${var}` inizia alla posizione offset.
- `${var:offset:len}` sottostringa di `${var}` di len caratteri, da offset. Ad esempio: `a="pippo", {a:2:2}=pp`
- `${#var}` lunghezza della stringa contenuta in var. Esempio: `a="pippo", ${#a}=5`
- `${var # pattern}` se var inizia con pattern, cancella il match più corto e ritorna la rimanente parte.
- `${var ## pattern}` se var inizia con pattern, cancella il match più lungo e ritorna la rimanente parte.
- `${var % pattern}` se var finisce con pattern, cancella il match più corto e ritorna la rimanente parte.
- `${var %% pattern}` se var finisce con pattern, cancella il match più lungo e ritorna la rimanente parte.

Facciamo un esempio:

```
# Sintassi:      stampaprimi file [n]
# stampa le prime n linee del file ordinato per valori decrescenti
#
filename=$1
filename=${filename:? "Manca il nome del file"}
howmany=${2:-5}
sort -nr $filename | head -${howmany}
```

ecco il nostro output:

```
bash> chmod u+x stampaprimi
bash> ./stampaprimi dati 3
3 Battisti
7 Pino Daniele
5 Bennato
bash> ./stampaprimi
stampaprimi: filename: Manca il nome del file
bash> ./stampaprimi 3
No such file
bash>
```

Controllo del flusso

- if/else
- for
- while
- until
- case
- select

Non vi spiegherò le piccolissime differenze di sintassi che esistono con gli altri linguaggi di programmazione anche perchè dovrei fare delle considerazioni e spiegazioni molto più approfondite. Magari in futuro tempo permettendo perfezionerò ed amplierò questa guida. Naturalmente per qualsiasi cosa mi potete contattare o utilizzare il blog ;)

Echo e Read

echo e read servono rispettivamente per stampare o leggere. Possiamo dire che sono come la printf e la scanf del C per capirci :) Ecco un esempio semplice:

```
bash> echo -n "Digita i valori di due variabili:"; read a b
Digita i valori di due variabili: pippo pluto
bash> echo $a
pippo
bash> echo $b
```

```
pluto  
bash>
```

Bene amici, ho finito almeno per il momento =) Qualsiasi cosa sono a disposizione e scusate la poca chiarezza, l'ho davvero scritta in fretta ho poco tempo ultimamente :°

Alla prossima!